AMRITA SCHOOL OF COMPUTING

DESIGN AND ANALYSIS OF ALGORITHMS

(23CSE211)

Name: N.Ashwath

Roll No.: CH.SC.U4CSE24133

Class: BTech (CSE-B)

School: Amrita School of Computing,

Chennai Campus.

LAB-4

1.MERGE SORT

```c
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];   // pivot element
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    quickSort(arr, 0, n - 1);
    printf("\nSorted array (Merge Sort): ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

OUTPUT:

```
n-ashwath@n-ashwath-VirtualBox:~$ ./a.out
Original array: 157 110 147 122 111 149 151 141 123 112 112 117 133
Sorted array (Merge Sort): 110 111 112 117 122 123 133 141 147 149 151  157
```

Time and space complexity :

Merge Sort always runs in **O(n log n) time** in the best, average, and worst cases because the algorithm always divides the array into two halves recursively (which gives **log n levels**) and at every level it performs a **merge step that processes all n elements**, leading to $n \times \log\ n$total work. However, its space complexity is **O(n)** because it requires an additional temporary array to store merged elements during the combining phase, meaning it is **not an in-place sorting algorithm** even though its recursion stack is small compared to the extra array memory

2.QUICK SORT:

```c
#include <stdio.h>
void merge(int arr[], int left, int mid, int right) {
    int i = left, j = mid + 1, k = 0;
    int size = right - left + 1;
    int temp[size];
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }
    while (i <= mid)
        temp[k++] = arr[i++];
    while (j <= right)
        temp[k++] = arr[j++];
    for (i = left, k = 0; i <= right; i++, k++)
        arr[i] = temp[k];
}
void mergeSort(int arr[], int left, int right) {
    if (left >= right)
        return;
    int mid = (left + right) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
int main() {
    int arr[] = {157,110,147,122,111,149,151,141,123,112,117,133};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    mergeSort(arr, 0, n - 1);
    printf("\nSorted array (Quick Sort): ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

OUTPUT:

```
n-ashwath@n-ashwath-VirtualBox:~$ ./a.out
Original array: 157 110 147 122 111 149 151 141 123 112 112 117 133
Sorted array (Quick Sort): 110 111 112 117 122 123 133 141 147 149 151 157
```

TIME and SPACE complexity:

Quick Sort has **O(n log n) time complexity** in the best and average cases because the pivot typically splits the array into roughly equal halves, giving **log n recursive levels** with **O(n) work per level**, but in the worst case (like when the pivot is always the smallest or largest element) the recursion becomes highly unbalanced and degrades to **$O(n^2)$**. Unlike Merge Sort, Quick Sort is **in-place**, so it does not use extra arrays; its space complexity is mainly due to the recursion stack, which is **O(log n) on average** for balanced partitions and can grow to **O(n)** in the worst case when the recursion becomes deep.